

Apache Bookkeeper: Un servizio di archiviazione scalabile, a tolleranza di errore e a bassa latenza

Hamado Dene

hamado.dene@studio.unibo.it

Marzo 2021

Contents

1	Obiettivo	4
2	Definizione di Apache Bookkeeper	4
3	Concetti e terminologia di Bookkeeper	4
3.1	Entries (records)	4
3.1.1	Aggiunta di entries	5
3.2	Ledgers	5
3.2.1	Ledger manager	6
3.3	Namespaces	7
3.4	Bookies	7
3.4.1	Gestione dei dati nei bookies	7
3.5	Data compaction	8
4	Archiviazione dei metadati	9
4.1	Architettura	9
4.1.1	Struttura	10
5	Perché Apache Bookkeeper?	10
5.1	Replication	11
5.2	Durability	12
5.3	Consistency	13
5.4	Availability	13
5.5	Low Latency	14
6	Analisi di un sistema basato su Bookkeeper HerdDB	15
6.1	Architettura di HerdDB	15
6.2	Replicazione dei dati	16
6.3	Ruolo di Bookkeeper nella leadership election	16
6.4	Log delle transazioni (Transaction log)	17
6.5	Analisi a basso livello	17
6.5.1	Installazione Zookeeper	17
6.5.2	Installazione HerdDB	19
6.6	Analisi Cluster	20
7	Installazione Bookkeeper	25
7.1	Installazione per dev	25
8	Interazione con Bookkeeper	26
8.1	Ledger API	26
8.2	Ledger Advanced API	33

9 DistributedLog (DL)	34
9.1 Logs	35
9.1.1 Log records	36
9.1.2 Log segments	36
9.1.3 Namespace	37
9.1.4 Writer	37
9.1.5 Reader	37
10 Sicurezza	37
10.1 TLS	38
10.2 Kerberos	39
10.2.1 Componenti	39
10.2.2 Protocollo	39
10.2.3 Configurazione kerberos su Bookkeeper	41
11 Demo	43
12 Conclusione	44

1 Obiettivo

Il progetto si pone come obbiettivo di studiare, analizzare e testare le principali caratteristiche di Apache Bookkeeper (<https://bookkeeper.apache.org/>). **Apache Bookkeeper** è un servizio che fornisce la memorizzazione persistente di flussi di log entries in sequenze chiamate ledgers. Queste entries vengono poi replicate in vari nodi di un sistema in cluster. E' stato progettato per essere affidabile e resistente ad una vasta gamma di guasti. I bookies si possono arrestare in modo anomalo, corrompere e scartare dati, ma finché ci sono abbastanza bookies che si comportano correttamente il servizio nel suo insieme continuerà a funzionare.

2 Definizione di Apache Bookkeeper

Apache Bookkeeper è un sistema di archiviazione progetto per fornire durabilità, coerenza e bassa latenza ottimizzato per carichi di lavoro in tempo reale. E' un servizio ampiamente distribuito per servire diversi casi d'uso, come la creazione di strutture ad alta disponibilità o di replica di sistemi distribuiti (es. HDFS namenode), fornendo la replica tra macchine in un singolo cluster e anche attraverso cluster di datacenter diversi (replica multi-datacenter).

3 Concetti e terminologia di Bookkeeper

Bookkeeper fornisce l'archiviazione replicata e durevole di flussi di log, ovvero flussi di records che formano una sequenza con un ordine ben definito.

3.1 Entries (records)

Una entry (noto anche come record o voce) contiene un'unità indivisibile di dati che un'applicazione client memorizza o legge da Bookkeeper. Questi dati vengono scritti all'interno dei ledgers (logs) come una sequenza di bytes. Quando un insieme di entries viene memorizzato in un ledger, ogni entry contiene i dati forniti dall'client e alcuni campi di metadati quali:

- **Ledger number:** L'id del ledger sulla quale è stata scritta la entry.
- **Entry number:** L'id che identifica la entry che è univoco all'interno del ledger.
- **Last confirmed(LC):** L'id dell'ultima entry che è stata registrata.
- **Data:** I dati presenti all'interno della entry, scritti dal client ovvero dall'applicazione.
- **Authentication code:** I codice di autenticazione del messaggio che viene utilizzato da BK per rilevare quando una voce è danneggiata o è stata manomessa.

I clients possono sempre iniziare a leggere da una entry particolare, o seguono la sequenza in tail, ovvero che rimangono in attesa del record successivo da aggiungere al registro. Possono anche ricevere i dati uno alla volta o in blocchi di record. Attraverso il ledger number è possibile recuperare anche le entry in maniera casuale.

3.1.1 Aggiunta di entries

Quando un client contatta un bookie per la scrittura di una entry in un ledger, vengono effettuati i seguenti passaggi:

- La entry viene aggiunta in sequenza all'entry log
- L'indice delle entry viene aggiornata nella ledger cache
- Viene aggiunta al journal una transazione corrispondente al tipo di aggiornamento richiesto dal client
- Viene mandato una risposta al client Bookkeeper

Per ragioni di performance, l'entry log memorizza le entries in memoria e li raggruppa in batch, mentre la ledger cache conserva le index pages in memoria e ogni tanto effettua un flush della cache. Il flush viene fatto solitamente in due casi:

- Quando viene raggiunto il limite di memoria della ledger cache, e quindi non c'è più spazio per contenere le index pages più recenti. Le entries meno recenti vengono quindi scritti in maniera persistente nell'index file di riferimento.
- Un thread sincrono in background periodicamente si occupa di effettuare un flush delle index pages dalla ledger cache alle index files.

3.2 Ledgers

I ledgers sono l'unità base di archiviazione in Bookkeeper, e contengono una sequenza di entries. Le entries in BK vengono scritti in sequenza e al massimo una sola volta, il che significa che i ledgers hanno una sequenza di solo aggiunta ovvero quando una entry viene scritta non può più essere modificata. Il ledger viene terminato quando un client lo chiude esplicitamente o quando un writer, un client che scrive una entry in esso si è bloccato. In questo caso Bookkeeper sigilla il ledger, e non sarà più possibile aggiungervi delle entries. La gestione dell'ordine di scrittura è responsabilità delle applicazioni client che hanno la possibilità di creare e di cancellare ledgers, leggere le entry e scrivere nuove entries all'interno dei ledgers. I ledgers possono essere utilizzati per memorizzare una sequenza di dati limitate (bounded sequences) e flussi illimitati (unbounded streams).

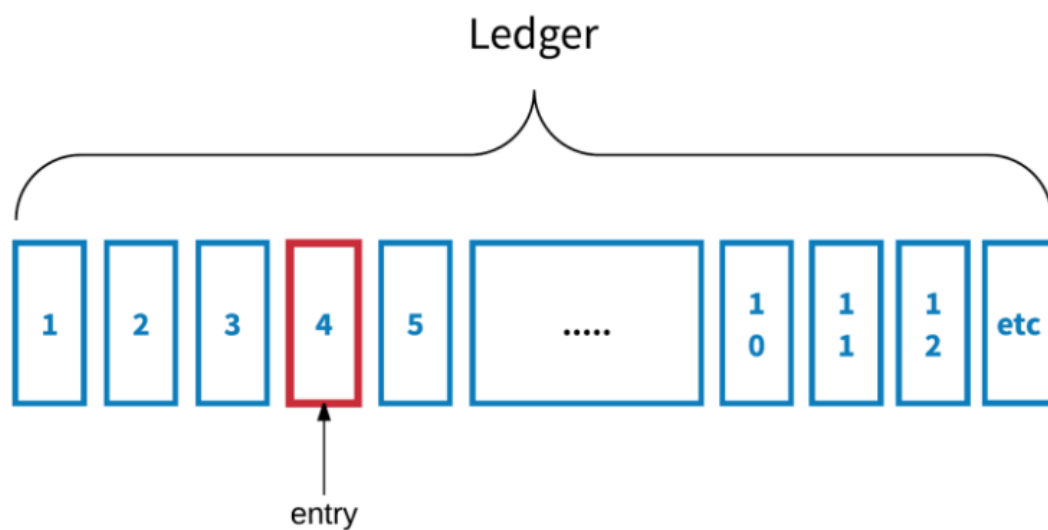


Figure 1: Ledger in Bookkeeper: bounded sequences

In Bookkeeper è possibile avere anche Uno stream (o log Stream o ledger Stream), che è una sequenza infinita di entry. A differenza di un ledger, è possibile aprirlo ogni volta per aggiungerci dei records. Uno Stream è semplicemente un insieme di ledgers, dove ogni ledger viene ruotato in base ad una politica di tempo o spazio. La politica principale di rotazione è il truncate, che consiste nella cancellazione dei ledgers più vecchi.

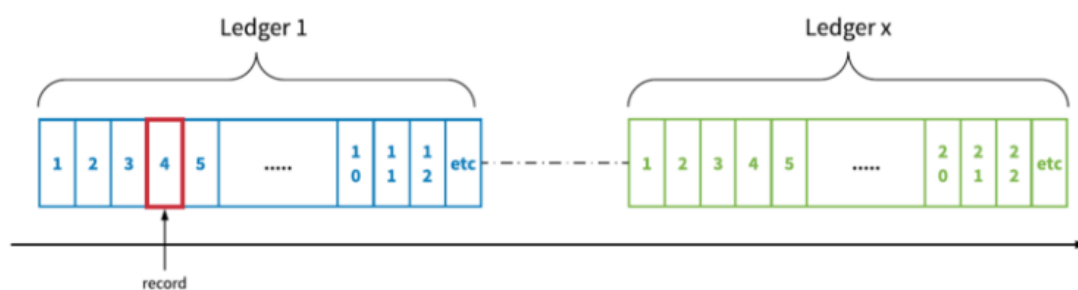


Figure 2: Ledger in Bookkeeper: unbounded stream

3.2.1 Ledger manager

La gestione dei metadati dei ledgers viene affidato al ledger manager. Esistono due tipi di ledger manager:

- **Flat ledger manager:** Memorizza i metadati dei registri in nodi figlio di un

singolo percorso Zookeeper. Crea dei nodi sequenziali per garantire l'unicità dell'ID del ledger.

- **Hierarchical ledger manager:** Ottiene innanzitutto un ID univoco globale da Zookeeper utilizzando un **Znode** di tipo **Ephemeral-sequential**. L'ID viene spezzato in 3 parti che costituiranno il percorso dove archiviare i metadati del ledger.

3.3 Namespaces

I log stream sono generalmente classificati e gestiti in un **namespace**. Un **namespace** è un meccanismo utilizzato dalle applicazioni per la gestione di streams. Tutti gli streams all'interno dello stesso namespace ereditano le stesse impostazioni e posizionano i record in base al placement policy configurato. Infatti un namespace è anche un'unità di distribuzione o amministrativa che consente di configurare un data placement policy a livello di namespace.

3.4 Bookies

I bookies sono singoli server Bookkeeper che gestiscono i ledgers (Più precisamente frammenti di ledgers). Un bookie è un singolo server di archiviazione e si occupa di conservare i frammenti di ledgers invece che interi ledgers (Questo per una questione di performance).

Per ogni Ledger L, un **ensemble** è un gruppo di bookies che hanno memorizzato l'entry L. Ogni volta che una entry viene scritta in un ledger, tale entry viene scritta in un sottogruppo di bookies anziché su tutti i bookies.

3.4.1 Gestione dei dati nei bookies

I bookie gestiscono i dati in modo strutturato per log, che è stato implementato basandosi su tre tipi di file:

- **Journals:** Un journal contiene i transaction log (log di transazione) di Bookkeeper. Prima di un qualsiasi aggiornamento ad un ledger, il bookie si assicura che la transazione sia scritta in uno storage non volatile. Viene quindi creato un file journal al l'avvio del bookie o quando il file journal precedente raggiungere la dimensione massima.
- **Entry logs:** Un entry log gestisce le entries ricevute dai client Bookkeeper. Le entries di ledgers differenti vengono aggregati e scritti in sequenza, mentre i loro offset vengono mantenuti come puntatori ad un ledger cache in modo tale da poter effettuare ricerche più rapide. L'entry log viene creato all'avvio del bookie o quando il file precedente raggiunge la dimensione massima configurata. I vecchi file log delle entries vengono rimossi dal thread del GC (garbage collector) una volta che non sono associati a nessun ledger attivo.

- **Index files:** Per ogni ledger viene creato un index file, che comprende un'intestazione e diverse pagine a lunghezza fissa che registrano gli offset dei dati memorizzati nelle entries logs. Questi files vengono aggiornati costantemente per garantire sempre buone prestazioni.

3.5 Data compaction

Come detto precedentemente, nei bookie, entries di ledger differenti vengono scritti in sequenza nelle entry logs. Per recuperare spazio, ogni bookie esegue un thread garbage collector per eliminare entry logs non associate. Se per esempio un determinato entry log contiene delle entries di un ledger che non è stato eliminato, entry log non verrà mai rimosso e lo spazio disco occupato non verrà mai recuperato. Per evitare questa casistica, il server bookie compatta le entry logs in un thread garbage collector per recuperare spazio disco.

Esistono due tipi di compattazione con frequenza diversa: La **Major compaction** e la **Minor compaction**. La differenza sta nel valore di soglia e intervallo di compattazione.

- La soglia di garbage collection è la percentuale di entry log occupati da ledgers non eliminati. Di default abbiamo 0,2 per la minor compaction e 0,8 per la major.
- L'intervallo di garbage collection è quanto frequentemente eseguire la compattazione. Di default abbiamo ogni ora per la minor e ogni 1 days per la major compaction.
- La compattazione si può disabilitare impostando a un valore minore o uguale a zero.

Il flusso per la compattazione è la seguente:

- Il thread analizza le entry logs per ottenere i loro metadati, che registra la lista dei ledgers compreso l'entry log e le percentuali corrispondenti.
- Con il normale flusso di garbage collection, dopo che il bookie ha determinato che il ledger è stato eliminato, il ledger viene rimosso dai metadata del entry log e la dimensione del entry log viene ridotto.
- Se la dimensione rimanente del entry log raggiunge una determinata soglia, le entries dei ledgers attivi nel entry log verranno copiate in un nuovo entry log.
- Una volta che le voci valide sono state copiate, il vecchio file di registro viene eliminato.

4 Archiviazione dei metadati

Bookkeeper richiede un servizio di archiviazione dei metadati per archiviare le informazioni relativi ai ledgers e ai bookies disponibili. Per gestire queste attività viene utilizzato **Apache Zookeeper**.

Apache Zookeeper è un servizio di coordinamento per applicazioni distribuite che consente la sincronizzazione di un cluster. Viene utilizzato per mantenere il sistema distribuito funzionante come se fosse una singola unità, attraverso le sue funzionalità di sincronizzazione, serializzazione e coordinamento. Può essere pensato come un file system in cui abbiamo degli znodes che memorizzano dati.

4.1 Architettura

Zookeeper funziona come un servizio di sincronizzazione replicato con eventual consistency. E' robusto, poiché i dati persistenti sono distribuiti tra più nodi (chiamati ensemble) e il client può connettersi ad un server specifico migrando se quel nodo fallisce. Vigge il concetto di leader, ovvero che un nodo leader viene scelto dinamicamente per consenso all'interno dell'ensemble; Se il nodo principale ha esito negativo, il ruolo di leadership passa ad un altro nodo.

Il leader è l'autorità per le scritture: In questo modo le scritture sono garantite come persistenti nell'ordine, ovvero sono lineari. Ogni volta che un client scrive nell'ensemble, i server collegati a quel client mantengono i dati insieme al leader. Pertanto la maggior parte dei server saranno sempre aggiornati sui dati. Tuttavia, questo rende impossibile eseguire scritture simultanee. La garanzia delle scritture lineari può essere problematica se Zookeeper viene utilizzato per un carico di lavoro dominante in scrittura. E' un servizio ideale per coordinare gli scambi di messaggi tra vari client, il che comporta minoranza di scritture a favore di maggior letture.

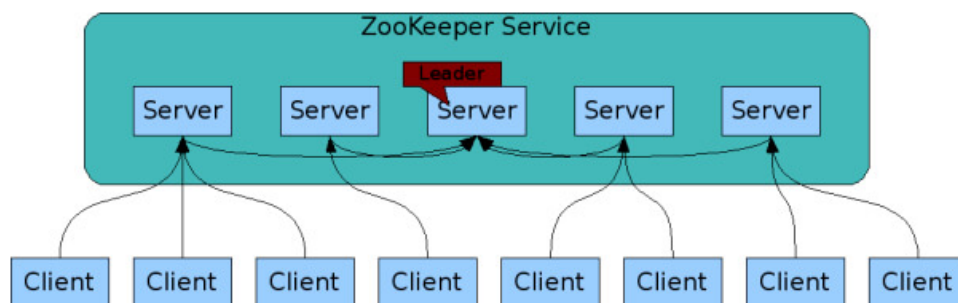


Figure 3: Zookeeper: Zookeeper architecture

4.1.1 Struttura

Il modello dei dati di Zookeeper è simile ad un file system, denominato **Znode**. L'unica differenza rispetto ad un file system è che i Znodes hanno la capacità di archiviare dati. Il loro compito è quello di mantenere le statistiche della struttura e i numeri di versione per le modifiche ai dati. Esistono vari tipi di Znodes quali:

- **Znodes persistenti:** E' il Znode predefinito di Zookeeper. Può essere eliminato solo chiamando la funzione di eliminazione di Zookeeper.
- **Znodes effimeri (Ephemeral Znode):** I dati vengono eliminati non appena la sessione viene chiusa o la connessione viene chiusa dal client.
- **Znodes sequenziali:** Ad un Znode sequenziale viene assegnato un intero unico crescente. Questo numero viene aggiunto al percorso utilizzato per creare lo Znode. Ovvero se il client crea uno Znode sequenziale con il percorso `"/x/x-`", Zookeeper assegna un numero di sequenza, ad esempio 1, e lo aggiunge al percorso. Avremo quindi `"/x/x-1"`. Questi tipi di Znode forniscono un modo semplice per creare Znode con nomi univoci. Offrono anche un modo più immediato per vedere l'ordine di creazione dei Znode.

Ogni volta che il client deve sapere se qualcosa è cambiato, accede al Znode. Essendo molto costoso in termini di risorse e latenza, per ridurre il polling, Zookeeper lavora sul modello di pubblica iscrizione, in cui ogni Znode pubblica delle **API** e il client può iscriversi. Ogni volta che viene modificato il contenuto del Znode, tutti i client iscritti a tale **API** riceveranno una notifica.

5 Perché Apache Bookkeeper?

Un sistema di archiviazione in tempo reale dovrebbe soddisfare tutti i seguenti requisiti:

- I client devono essere in grado di scrivere e leggere flussi di entry con una latenza molto bassa (inferiore ai 5ms).
- L'archiviazione dei dati deve essere durevole, coerente e tollerante ai guasti.
- Il sistema deve permettere ai client di eseguire lo stream o tail dei ledgers per propagare i dati man mano che vengono scritti.
- Il sistema deve essere in grado di archiviare e fornire l'accesso ai dati storici in tempo reale.

Bookkeeper soddisfa tutti questi requisiti fornendo **Replication, Durability, Consistency, Availability, Low Latency**.

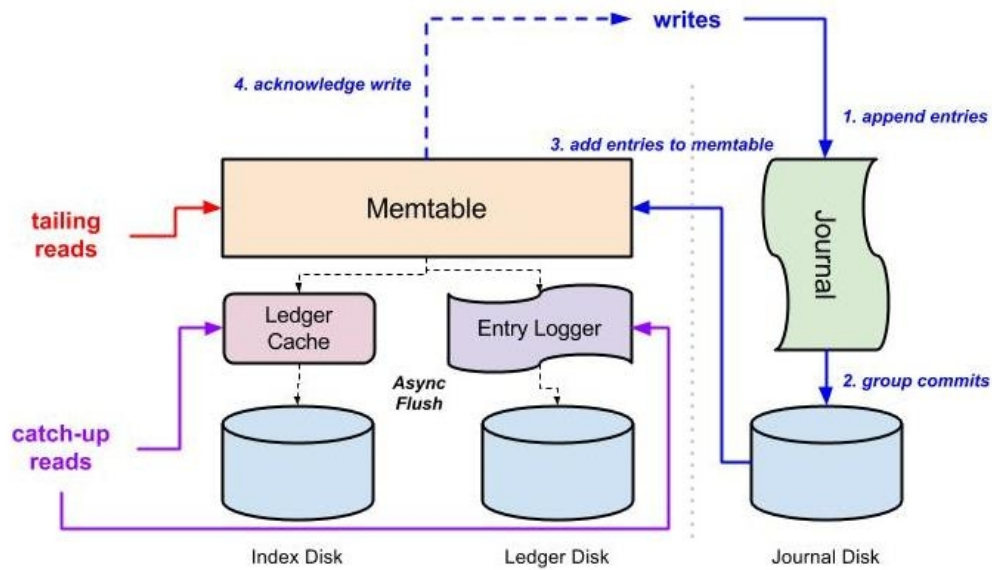


Figure 4: Architettura I/O di BookKeeper

5.1 Replication

Bookkeeper replica e archivia diverse copie identiche di ciascun record di dati, su macchine diverse all'interno di un datacenter o su più datacenter. A differenza di altri sistemi basati sulla replica master/slave, Bookkeeper utilizza un algoritmo di replica parallela basato sul quorum-vote per replicare i dati e garantire una bassa latenza.

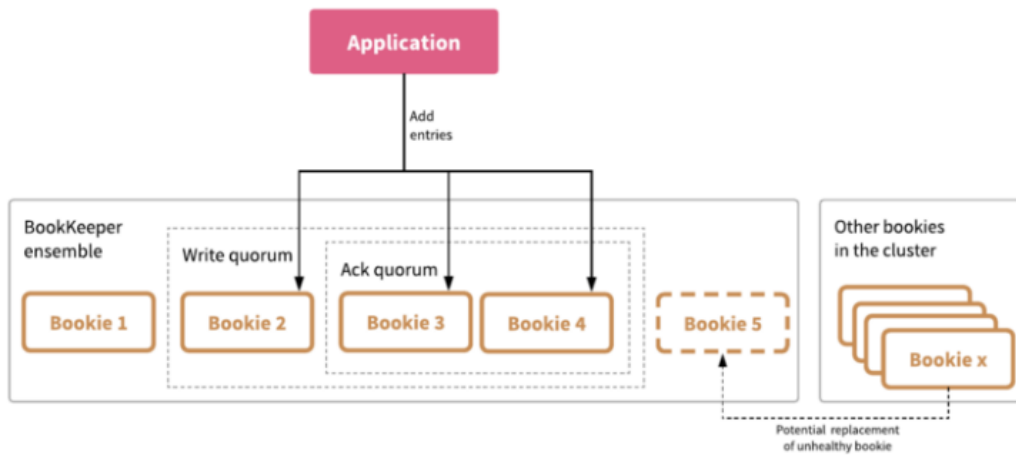


Figure 5: Apache Bookkeeper: Ensemble, write, and ack quorums in BookKeeper replication

Al momento della creazione di un ledger, vengono settati tre parametri:

- **Ensemble size (ES):** Indico il numero di bookie in cui devono essere memorizzati i dati dei ledgers.
- **Write quorum size (WQ):** Il numero di copie che devono essere effettuate per ogni Entry.
- **Ack quorum size (AQ):** Il numero richiesto di ack utili per considerare la scrittura conclusa con successo.

Supponiamo di avere 3 macchine. La configurazione ideale sarebbe avere $ES=2$, $WQ=2$, $AQ=2$ (Se si hanno pochi bookie è la configurazione migliore. es 3 bookies). Avremo quindi che ogni scrittura viene replicato due volte (Ovvero scritto su almeno due bookie), e per considerare successo la scrittura il client deve ricevere almeno 2 ack. Avere ES maggiore di WQ aiuta a migliorare le prestazioni poiché permette di diffondere le scritture e le letture su più bookies. Considerando la nostra configurazione di esempio, se un nodo viene perso avremo comunque due nodi disponibili su cui replicare i dati. In caso di fallimento di un bookie lo scrittore (writer) sceglie un nuovo ensemble (detto anche cambio di gruppo) su cui scrivere i dati, in maniera totalmente trasparente e finché ci sono abbastanza bookies il sistema continuerà a funzionare correttamente. Ad ogni cambio gruppo viene iniziato un nuovo segmento di ledger, e i lettori in base ai cambiamenti sui metadati sono in grado di connettersi automaticamente ai nuovi bookies.

5.2 Durability

Ogni record scritto su Bookkeeper è garantito per essere replicato e archiviato in modo durevole nel numero configurato di Bookie. Questo si ottiene utilizzando **fsync** esplicito

e la conferma della scrittura. La funzione **fsync** ha lo scopo di forzare una scrittura fisica dei dati dalla cache del buffer e di assicurare che, dopo un crash del sistema o un altro errore, tutti i dati fino al momento della chiamata **fsync** vengano registrati sul disco.

- Su un singolo Bookie, i dati vengono scritti esplicitamente (**fsynced**) sul disco prima che gli **ack** vengano inviati ai clients, al fine di rendere persistenti i dati in caso di crash. Ciò garantisce che i dati vengano scritti su una memoria persistente che non richiede alimentazione, in una forma che ne consente il ripristino e l'utilizzo.
- All'interno di un cluster, i records di dati vengono replicati su più **Bookie** per la tolleranza agli errori.
- Un record di dati viene riconosciuto solo quando i client ricevono un numero configurato di risposte (Specificato dalla dimensione del quorum di **ack**) dai bookies.

In alcuni sistemi distribuiti (come database distribuiti o file system distribuiti) presumono che la semplice replica dei dati su più nodi nello storage/memoria sia sufficiente per soddisfare i requisiti di durabilità, ma questi presentano un potenziale problema di perdita dei dati che ormai risultano fondamentali nei business aziendali. Bookkeeper garantisce una durability molto elevata, con l'obiettivo di prevenire completamente la perdita di dati.

5.3 Consistency

Garantire la consistency è uno dei problemi più comuni nei sistemi distribuiti, soprattutto quando viene introdotta la replica per fornire durability e availability. Bookkeeper garantisce una consistency semplice ma forte (repeatable read consistency) per i dati archiviati nei log:

- Se un record è stato riconosciuto dall'applicazione, deve essere immediatamente leggibile.
- Se un record viene letto una volta, deve essere sempre leggibile.
- Se una scrittura del record **R** ha dato esito positivo, tutti i record fino a **R** vengono salvati/mantenuti correttamente e saranno sempre leggibili.
- L'ordine dei record memorizzati deve essere esattamente identico tra i lettori e ripetibile.

Questa repeatable read consistency viene ottenuta da un protocollo **LastAddConfirmed (LAC)** utilizzato da Bookkeeper.

5.4 Availability

Nel contesto del teorema CAP (Consistency, Availability, Partition tolerance), Bookkeeper è un sistema **CP**. Tuttavia, Bookkeeper offre una disponibilità molto elevata anche in presenza di hardware, rete e altri guasti. Per garantire un'elevata disponibilità in lettura e scrittura, Bookkeeper utilizza i meccanismi di:

- **Write availability:** Quando si verifica un crash di un bookie, i client dovranno cambiare il bookie sulla quale scrivono i records. Ciò garantisce che le scritture avranno sempre esito positivo fintanto che nel cluster rimane un numero sufficiente di bookie totali.
- **Read availability:** A differenza di altri sistemi che designano un nodo come leader, Bookkeeper permette ai clients di leggere i records da qualunque bookie nel ensemble. Questo permette di propagare il carico tra i vari bookies, permettendo di ridurre la latenza in lettura e in tail.

5.5 Low Latency

Garantire durability, consistency sono problemi molto complessi in un sistema distribuito, soprattutto se combinati con l'obiettivo di soddisfare requisiti di bassa latenza. Bookkeeper per soddisfare questi requisiti, utilizza i seguenti modi:

- Su un singolo Bookie, il server bookie è progettato per l'isolamento **I/O** tra diversi carichi di lavoro (scrittura, lettura in tail ,catch-up/lettura casuale).
 - **Scritture:** Le scritture vengono eseguite in un file journal e il gruppo viene committato in un disco journal dedicato. Una volta eseguito il commit, i dati vengono aggiunti a una tabella in memoria (memtable) e una risposta viene inviato al cliente. Ciò significa che il percorso di scrittura è un I/O del disco puramente sequenziale ed è isolato dai carichi di lavoro di lettura.
 - **Lettura in tail:** I nuovi records vengono aggiunti in una memtable e quando la memtable è piena, i dati vengono riversati nella memoria del ledger. Pertanto la maggior parte delle tailing read, leggerà direttamente i dati da un memtable e non sarà influenzata dalle Catch-up reads. Anche se il memtable viene riportato nel filesystem , i dati sono comunque disponibili nella cache del filesystem.
 - **Catch-up reads:** Le letture in questo caso vengono fornite direttamente dall'archivio del ledger che è composto da files di entry indicizzate. I dati vengono intercalati e archiviati in batch in questi files, il che consente operazioni di scrittura sequenziali di grandi dimensioni e migliora la località dei dati per le letture. Gli indici sono abbastanza piccoli da stare in una cache di memoria, riducendo gli accessi casuali su disco e migliorando ulteriormente l'efficienza delle catch-up reads (letture di recupero).
- Per mascherare la penalità di latenza derivante da errori di rete, pause di Garbage collection JVM e dischi lenti, viene utilizzato uno schema di replica parallela basato sul voto del quorum (quorum-vote parallel replication scheme).
- Un meccanismo di long-pooling viene utilizzato per notificare e consegnare i records scritti di recente ai tailing readers non appena vengono riconosciuti e confermati.

6 Analisi di un sistema basato su Bookkeeper HerdDB

HerdDB è un database distribuito, dove i dati vengono replicati in un cluster di server (Ovvero in un sistema distribuito) senza la necessità di uno storage condiviso. La sua architettura è basata sul concetto di database NoSQL, quindi a basso livello è semplicemente un database di tipo chiave-valore con una parte di astrazione SQL, che consente all'utente di sfruttare le sue attuali conoscenze e trasferire applicazioni già esistenti su HerdDB. L'obiettivo di HerdDB è quello di fornire un'applicazione che sia ottimizzata per "Scritture" veloci con schemi di lettura (Read) e aggiornamenti (Update) basati sulla chiave primaria.

6.1 Architettura di HerdDB

Per capire come viene sfruttato Bookkeeper su herddb, abbiamo bisogno di capire l'architettura di herddb e come sono organizzati i dati all'interno di esso.

I dati all'interno di HerdDB sono organizzati sotto forma di tabelle come in qualsiasi database SQL. Essendo un database distribuito, i dati vengono replicati usando il concetto di replicazione dei dati e le tabelle sono organizzate all'interno di tablespaces.

Un tablespace è un insieme logico di tabelle ed è il concetto architettonico fondamentale su cui è costruita la replica dei dati. Qui entra in gioco Bookkeeper che fornisce a HerdDB un registro distribuito su cui scrivere le [transazioni](#). Le caratteristiche principali sono:

- Ogni scrittura è garantita come durevole dopo aver ricevuto l'ACK da parte di Bookkeeper.
- Viene garantito che ogni replica legga solo le **entries** per la quale l'ACK è stato ricevuto.
- Ogni **ledger** (l'unità di memoria di base di Bookkeeper) può essere scritto una sola volta.



Figure 6: HerdDB: Architettura

6.2 Replicazione dei dati

In installazione di tipo cluster, per ogni tablespace viene delegato un nodo leader e un numero di nodi come repliche, che vengono denominati follower. Il leader è l'unico che può accettare query, ed è quindi l'unico nodo in grado di modificare i dati di **transazione** e i metadati dei tablespaces. In altri termini è l'unico nodo che può effettuare manipolazioni ai dati come insert/update.

Per ogni mutazione di dati/metadati, il leader del tablespace scrive una voce sul **log delle repliche** ([transaction log](#)) basato su (Apache Bookkeeper) e poi aggiorna successivamente la copia locale dei dati. I nodi follower, rimangono sempre in tail sul log delle repliche e aggiornano continuamente la loro copia locale dei dati.

6.3 Ruolo di Bookkeeper nella leadership election

Il ruolo di leader è deciso in maniera abbastanza statico, quindi non ci sono elezioni continue. E' possibile per l'amministratore di sistema settare un nodo leader, attraverso un parametro di configurazione e decidere per ogni tablespace il numero minimo di repliche che deve avere (ovvero su quanti nodi questo tablespace dovrà essere replicato). Se il leader diventa irraggiungibile, dopo un tempo T configurabile dell'amministratore viene nominato leader uno dei nodi follower tramite un algoritmo di leadership election.

Quando un nodo parte come leader, deve assicurarsi di avere abbastanza dati. In un sistema distribuito di questo tipo, il follower sarà sempre leggermente indietro rispetto al nodo leader (questo può essere anche causato dalla latenza di rete). Bookkeeper però ci

garantisce che quando viene scritto una entry, quest'ultima viene considerata consistente solo se viene ricevuto l'ACK dal numero adeguato di nodi configurati. Questo ci garantisce che quando il nodo follower parte, avrà comunque la possibilità di andare a scaricarsi i dati mancanti da Bookkeeper, avendo la garanzia che quei dati sono consistenti.

6.4 Log delle transazioni (Transaction log)

Una transazione indica una qualunque sequenza di operazioni lecite che, se eseguita in modo corretto, produce una variazione nello stato di una base di dati. In caso di successo, il risultato delle operazioni deve essere permanente o persistente, mentre in caso di insuccesso si deve tornare allo stato precedente all'inizio della transazione.

Il nucleo di HerdDB è il log delle transazioni, che è basato su Apache Bookkeeper. In caso di replica, ogni mutazione sui dati viene scritta sul transaction log, e quindi applicata alla versione in memoria della riga corrispondente. Questo registro viene continuamente letto dai follower, in questo modo ogni replica applica le stesse modifiche ai dati e rimane sincronizzata con il suo leader. Ad ogni scrittura sul log viene assegnato un numero progressivo che sarà la chiave. Con questa chiave i dati possono essere trasferiti in maniera coerente sulla rete.

6.5 Analisi a basso livello

Per riuscire ad effettuare un'analisi a basso livello, abbiamo bisogno di un cluster di macchine su cui andare a installarci un cluster zookeeper e un cluster herdDB. Non avremo bisogno di un cluster Bookkeeper in quanto herdDB fa partire nello stesso processo un cluster Bookkeeper.

Per il cluster di macchine useremo 3 macchine virtuali che ci permetteranno di eseguire l'installazione.

6.5.1 Installazione Zookeeper

Il secondo step è l'installazione e la configurazione del cluster zookeeper.

Tramite il sito ufficiale di [zookeeper](https://zookeeper.apache.org/) scarichiamo l'ultima release e lo spaccettiamo nei 3 server nello stesso percorso in modo da rendere più facile la configurazione.

Su ognuna delle 3 macchine dovremo eseguire i seguenti comandi:

```
- mkdir /data/herddb-cluster/  
- cd /data/herddb-cluster  
- wget https://zookeeper-latest-stable-release  
- tar -xvzf zookeeper-latest-stable-release.tar.gz  
- ln -s zookeeper-latest-stable-release zookeeper
```

```
- cd /data/herddb-cluster/zookeeper/conf
- cp zoo_sample.cfg zoo.cfg
```

Tutti i nodi che formeranno il quorum zookeeper avranno bisogno dello stesso file di configurazione. Nel file zoo.cfg di ciascuno dei 3 nodi, dobbiamo modificare la configurazione come la seguente:

```
tickTime=2000
dataDir=/data/herddb-cluster/zookeeper
clientPort=2181
maxClientCnxns=60
initLimit=10
syncLimit=5
server.1=your_zookeeper_node_1_ip:2888:3888
server.2=your_zookeeper_node_2_ip:2888:3888
server.3=your_zookeeper_node_3_ip:2888:3888
```

- **ticktime:** Utilizzato come heartbeats per tutte le operazioni dipendenti dal tempo.
- **DataDir:** Indica il path dove memorizzare i dati di zookeeper.
- **clientPort:** Indica la porta che potrà usata dai client per interagire con zookeeper (2181 di default).
- **maxClientCnxns:** Indica il numero massimo di client che possono essere connessi simultaneamente.
- **initLimit:** Indica quanti tick attendere durante i tentativi di connessione al nodo zookeeper. Se impostato per esempio a 10 e il ticketTime a 2000, allora un nodo zookeeper attende 10*2000 ms prima di rinunciare alla connessione e decidere di eleggere un nuovo leader.
- **syncLimit:** Indica il tempo che può trascorrere tra l'invio di una richiesta e la ricezione di un riconoscimento. E' il tempo massimo in cui i nodi possono non essere sincronizzati con il leader.
- **server.x:** Indicano per ogni nodo quali porte usare per la comunicazione tra i nodi di zookeeper. Una porta per i collegamenti dei follower (nel nostro caso 2888) se leader e una porta per altre connessioni durante la fase di elezione del leader (Nel nostro caso 3888).

Per completare la configurazione del cluster zookeeper, dobbiamo specificare un **ID** unico su ciascuno dei server. All'interno del dataDir, andremo a creare un file **myid** che specificherà l'id del nodo. L'ID dovrà corrispondere al numero del server specificato in zoo.cfg. quindi se per un server abbiamo specificato server.1 allora dovrà avere come id 1.

```

#Per il nodo 1
- echo 1 > /data/herddb-cluster/zookeeper/myid
#Per il nodo 2
- echo 2 > /data/herddb-cluster/zookeeper/myid
#Per il nodo 3
- echo 3 > /data/herddb-cluster/zookeeper/myid

```

Ora possiamo avviare il servizio tramite il comando di start

```

- /data/herddb-cluster/zookeeper/bin/zkServer.sh start
↪ /data/herddb-cluster/zookeeper/conf/zoo.cfg

```

Se i servizi partono correttamente nei logs dovremo vedere un nodo leader e gli altri due follower:

```

#Nodo 1 - Follower
2021-05-13 15:14:07,751 [myid:1] - INFO
↪ [WorkerReceiver[myid=1]:FastLeaderElection$Messenger$WorkerReceiver@390]
↪ - Notification: my state:FOLLOWING; n.sid:3, n.state:LOOKING,
↪ n.leader:3, n.round:0x1, n.peerEpoch:0x1, n.zxid:0x0, mess
----
#Nodo 2 - Leader
2021-05-13 21:14:07,756 [myid:2] - INFO
↪ [WorkerReceiver[myid=2]:FastLeaderElection$Messenger$WorkerReceiver@390]
↪ - Notification: my state:LEADING; n.sid:3, n.state:LOOKING,
↪ n.leader:3, n.round:0x1, n.peerEpoch:0x1, n.zxid:0x0, message format
↪ version:0x2, n.config version:0x
---
#Nodo 3 - Follower
2021-05-13 15:14:07,759 [myid:3] - INFO
↪ [WorkerReceiver[myid=3]:FastLeaderElection$Messenger$WorkerReceiver@390]
↪ - Notification: my state:LOOKING; n.sid:1, n.state:FOLLOWING,
↪ n.leader:2, n.round:0x1, n.peerEpoch:0x2, n.zxid:0x0, message format
↪ version:0x2, n.config version:0x0

```

6.5.2 Installazione HerdDB

Ora che abbiamo completato l'installazione di un cluster **Zookeeper**, possiamo procedere con l'installazione di Herddb. Tramite github possiamo scaricare l'ultima versione stabile di [Herddb](#). Su ognuna delle tre macchine eseguire i seguenti comandi:

```

- cd /data/herddb-cluster
- wget https://herddb-latest-stable-release
- unzip herddb-latest-stable-release.zip
- ln -s herddb-latest-stable-release herddb
- cd /data/herddb-cluster/herddb/conf

```

Herddb può essere installato sia in versione standalone (Quindi con i log direttamente su disco) o in modalità cluster con i log su Bookkeeper. Nel nostro caso andremo a modificare la configurazione in modo tale che il servizio parta in modalità cluster. Attraverso il file **server.properties** andiamo a fare il setup di Herddb per la modalità cluster:

```
server.mode=cluster
#per host lo mettiamo in ascolto su tutte le interface
server.host=0.0.0.0
server.zookeeper.address=your_zookeeper_node_1_ip:2181,
                        your_zookeeper_node_2_ip:2181,
                        your_zookeeper_node_3_ip:2181
```

Una parte molto importante della configurazione di Herddb riguarda la replicazione. In modalità cluster, il tablespace può essere replicato su più nodi, ovvero che un solo tablespace può avere più copie in base alla replica count configurato. Di default quando viene creato un tablespace, quest'ultimo parte con replica 1, quindi sarà presente solo una copia di quel tablespace. Avendo 3 nodi, l'ideale è avere due copie per ogni tablespace, così da avere sempre un nodo disponibile qualora se ne perdesse uno. Questo ci permette di avere una tolleranza ai guasti.

```
tablespace.default.replica.count=2
```

La configurazione di questo parametro non è sufficiente però ad avere una forte tolleranza ma per averlo dobbiamo avere almeno un quorum size lato Bookkeeper che determina il numero di copie garantite del commit log.

```
server.bookkeeper.ensemble.size=2
server.bookkeeper.write.quorum.size=2
server.bookkeeper.ack.quorum.size=2
```

Avviare il servizio tramite il comando:

```
- bin/service server start
```

6.6 Analisi Cluster

Ora che il cluster Herddb è installato correttamente possiamo andare ad analizzare il modo in cui viene gestito la replicazione dei dati. Per fare ciò è possibile utilizzare la CLI che Herddb mette a disposizione, la cli di bookkeeper e un software pensato per il management di bookkeeper in maniera visuale che è [bookkeeper visual manager](#).

Andando ad analizzare **bookkeeper visual manager**:

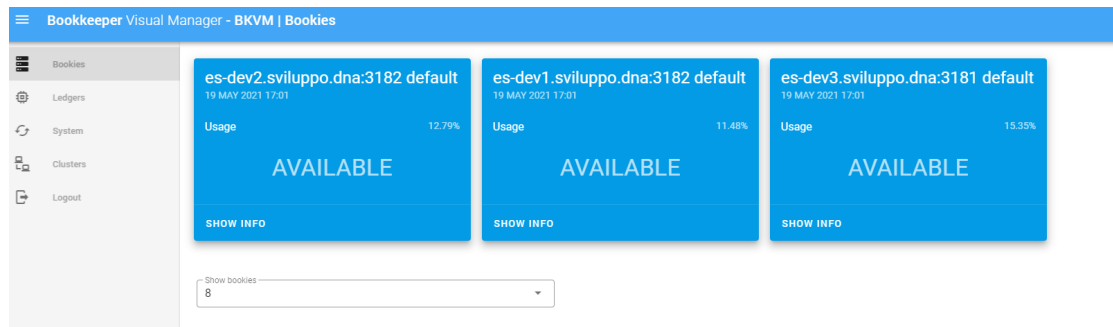


Figure 7: Bookkeeper: Cluster Bookkeeper embedded in herddb

Nella figura 6, vediamo i 3 **bookie** che girano nello stesso processo di **Herddb**. Ovvero ogni nodo di **Herddb** durante la sua inizializzazione fa partire un **bookkeeper embedded**. Nel nostro caso abbiamo 3 nodi (es-dev1.sviluppo.dna:, es-dev2.sviluppo.dna:, es-dev3.sviluppo.dna:).

Se andiamo nel tab **ledger** vedremo la lista dei ledger che sono stati creati da Herddb.

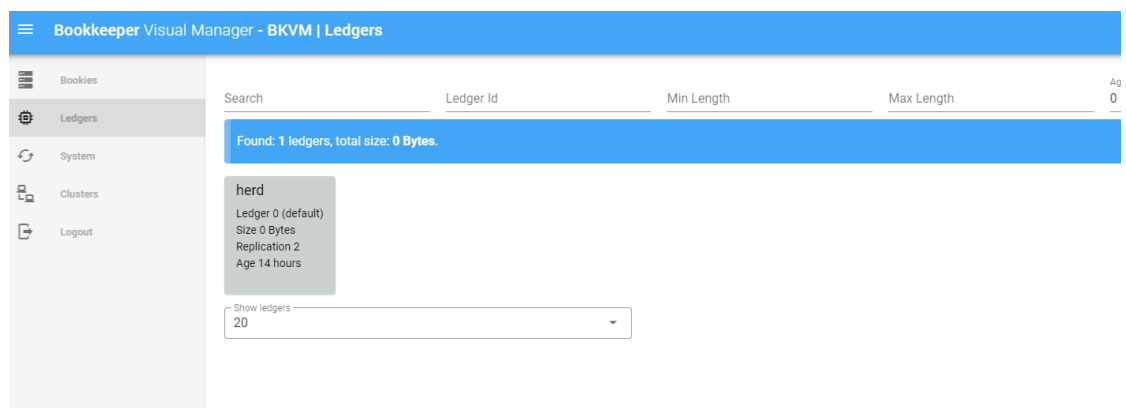


Figure 8: Bookkeeper: Cluster Bookkeeper ledgers list

Andando ad analizzare nel dettaglio un ledger possiamo vedere:

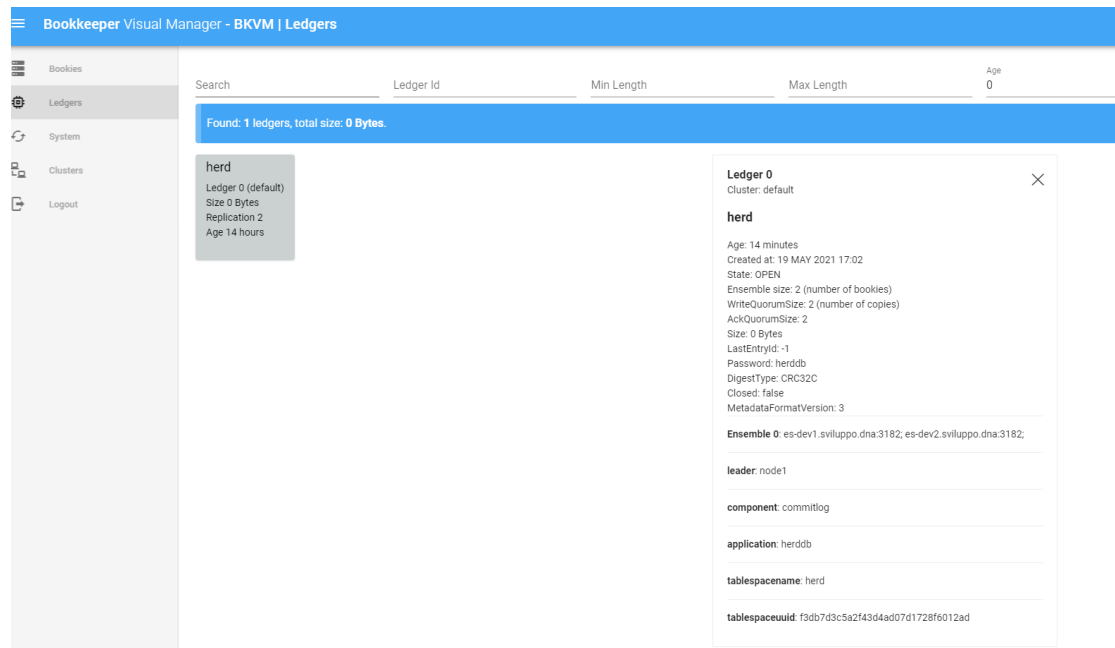


Figure 9: Bookkeeper: Cluster Bookkeeper ledger detail

- **State: OPEN** —>indica lo stato del ledger. Può essere in stato **OPEN** o **CLOSED**
- **Ensemble size: 2 (number of bookies)** —>Indica il numero di bookies che devono essere considerati per formare ensemble (Preso dalla configurazione fatto in **server.properties** di Herddb). In questo caso l'ensemble è formato da **Ensemble 0: es-dev1.sviluppo.dna:3182; es-dev2.sviluppo.dna:3182;**
- **WriteQuorumSize: 2 (number of copies)**—>Indica il numero di copie su cui replicare i dati
- **AckQuorumSize: 2** Per considera valida la scrittura di una entry, ci sarà bisogno almeno di 2 **ACK** da parte dell'ensemble.
- **leader: node1** —>Questo parametro indica il nodo leader del tablespace, quindi parliamo in questo caso di Herddb. Ogni ledger sarà associato ad uno specifico tablespace che avrà solo un nodo leader in grado di scrivere su quello specifico ledger.
- **LastEntryId: -1** —>Indica l'ID dell'ultima entry che è stato scritto all'interno del ledger. Se non c'è nessuna entry all'interno del ledger, allora ID è settato a -1.

Ora proviamo a creare un tabella all'interno del tablespace herd e inseriamoci un dato. Prima di tutto dobbiamo connetterci alla CLI tramite il comando:

```
bin/herddb-cli.sh -x jdbc:herddb:zookeeper:zookeeper_ip:2181/herd -sc
```

```
herddb(herd): select * from systablespace
```

tablespace_name	uuid	leader	replica	expectedreplicacount	maxleaderinactivitytime
herd	2453046e66f64884bc240359efeeb973	node1	node2,node1	2	0

Figure 10: Herddb: List of tablespaces

Guardando il tablespace **herdb** vediamo che il campo replica indica i nodi dove sono memorizzati di dati di questo tablespace. Uno dei nodi viene considerato leader, ovvero che sarà quello che potrà agire sul tablespace per ogni mutazione dei dati. La variabile **expectedreplicationcount** indica il numero di repliche atteso per quel tablespace, Tenendo conto anche del numero di ensemble disponibili.

```
herddb(herd): select * from systables
```

tablespace	table_name	table_uuid	systemtable
herd	dual	f0e9eb5c-47b5-4eb7-a808-3426ba750c98	true
herd	syscolumns	1306627f-73be-4ed0-be27-51fde90e3242	true
herd	sysforeignkeys	d4c4f767-a6f0-4ce9-a0ba-4e9eca6d9590	true
herd	sysindexcolumns	52445346-afbd-498a-a15c-4591985d103c	true
herd	sysindexes	4e689eb4-f8fd-4067-9ef5-37a24955a22a	true
herd	syslogstatus	9575dc5f-7e86-4ac0-a427-b3cfa9509a45	true
herd	sysnodes	80998ea6-687e-4279-9b29-ff4535f97ce0	true
herd	sysstatements	3cca777a-c50b-49bc-9bf4-8c1916beee02	true
herd	systables	e2fa6ebc-36d5-4ee2-a0f8-16bff4d09bf0	true
herd	systablespace replicastate	59346f04-4028-42d6-9e19-eb450f0006ae	true
herd	systablespace	4fd1a006-bc21-4008-82f6-d9dd7cc65a23	true
herd	systablestats	3555c86e-7ca3-46e6-bf0b-6de4a2ab0e3b	true
herd	systransactions	7457db89-8e47-46da-8a09-flada390c769	true

Figure 11: Herddb: List of tables inside tablespace herd

```
#Create a table
herddb(herd): CREATE TABLE Persons(PersonID int,LastName
↪ varchar(255),FirstName varchar(255))
#Insert into a tables
INSERT INTO Persons (PersonID, LastName, FirstName) VALUES (1,'Tom B.
↪ Erichsen','Skagen 21')
```

```
herddb(herd): INSERT INTO Persons (PersonID, LastName, FirstName) VALUES (1,'Tom B. Erichsen','Skagen 21')
herddb(herd): select * from persons
```

personid	lastname	firstname	_pk
1	Tom B. Erichsen	Skagen 21	1

Figure 12: Herddb: Table persons data

Ogni volta che si inserisce un dato nella tabella è come se andassimo a scrivere una **entry** all'interno del ledger in stato OPEN di quel tablespace. Se attraverso la CLI di bookkeeper andiamo ad analizzare ledger 0, vedremo che al suo interno ci sono una serie di entry.

```
[es@es-dev1 bookkeeper-server-4.13.0]$ bin/bookkeeper shell ledger -m 0
17:55:36,374 INFO openFileLimit = 20000
17:55:36,420 INFO maxDirectMemory = 1073741824, pageSize = 8192, pageLimit = 43690
17:55:36,450 INFO Over Replicated Ledger Deletion : enabled=true, interval=86400000
17:55:36,463 INFO Minor Compaction : enabled=true, threshold=0.20000000298023224, interval=3600000
17:55:36,463 INFO Major Compaction : enabled=true, threshold=0.800000011920929, interval=86400000
===== LEDGER: 0 =====
master key : dcb51477a66d9ab5fb35f9bd03add16bee873ad9
size       : 8192
entries    : 1024
isFenced   : false
===== LEDGER: 0 =====
entry 0 : (log:0, pos: 1028)
entry 1 : (log:0, pos: 1088)
entry 2 : (log:0, pos: 1275)
entry 3 : (log:0, pos: 1335)
entry 4 : (log:0, pos: 1449)
entry 5 : (log:0, pos: 1509)
entry 6 : (log:0, pos: 1569)
entry 7 : (log:0, pos: 1629)
entry 8 : (log:0, pos: 1689)
entry 9 : (log:0, pos: 1749)
entry 10 : (log:0, pos: 1809)
entry 11 : (log:0, pos: 1869)
entry 12 : (log:0, pos: 1929)
entry 13 : (log:0, pos: 1989)
entry 14 : (log:0, pos: 2049)
entry 15 : (log:0, pos: 2109)
entry 16 : (log:0, pos: 2169)
```

Figure 13: Bookkeeper: Ledger 0 part of entries

Se andiamo ad analizzare più nel dettaglio le entry, possiamo vedere la **CREATE TABLE** e **INSERT** che abbiamo fatto:

```
[es@es-dev1 bookkeeper-cli]$ bin/bookkeeper shell readlog -msg 0 -ledgerid 0 -entryid 1
17:10:35,623 INFO Scan entry log 0 (0.log) for LedgerId 0 for EntryId 1
17:10:35,793 INFO ----- Lid=0, Eid=1, ByteOffset=1084, EntrySize=183 -----
17:10:35,793 INFO Type: DATA
17:10:35,793 INFO LastConfirmed: 0
17:10:35,793 INFO Data:
17:10:35,793 INFO
zQc
y{P~personsuherdpersons$85771faa-117b-4f84-b91b-ff6a580b2b5a_ppersonilastname firstname_pk
17:10:35,808 INFO
[es@es-dev1 bookkeeper-cli]$
```

Figure 14: Bookkeeper: Ledger 0 CREATE TABLE entry

```
[es@es-dev1 bookkeeper-cli]$ bin/bookkeeper shell readlog -msg 0 -ledgerid 0 -entryid 2
17:11:23,444 INFO Scan entry log 0 (0.log) for LedgerId 0 for EntryId 2
17:11:23,625 INFO ----- Lid=0, Eid=2, ByteOffset=1271, EntrySize=110 -----
17:11:23,625 INFO Type: DATA
17:11:23,625 INFO LastConfirmed: 1
17:11:23,625 INFO Data:
17:11:23,625 INFO
17:11:23,625 INFO {"person": "Tom B. Erichsen", "skagen": 21}
17:11:23,634 INFO
[es@es-dev1 bookkeeper-cli]$
```

Figure 15: Bookkeeper: Ledger 0 INSERT entry

7 Installazione Bookkeeper

Una tipica installazione di Bookkeeper è costituita da un insieme di [bookies](#) e da un quorum **Zookeeper**. Il numero esatto di bookie dipende dalla modalità di quorum che si vuole implementare, dalla velocità effettiva desiderata e dal numero di client che utilizzano contemporaneamente l'installazione. Per parlare di un sistema distribuito abbiamo bisogno di avere almeno 3 bookies. Non esiste chiaramente un limite al numero di bookie che si può gestire in un ensemble.

A livello hardware, bookkeeper consiglia di avere almeno due dischi di cui uno viene utilizzato per i dati del **journalNode** e il secondo disco per memorizzare i **ledger**. E' possibile però eseguirlo anche su un disco, ma questo però comporta prestazioni inferiori.

Per quanto riguarda invece Zookeeper non ci sono vincoli sul numero di nodi da configurare. Basterebbe anche una semplice installazione standalone, ma per una resilienza superiore è consigliato avere almeno 1 cluster da almeno 3 macchine.

Come abbiamo visto nell'installazione di [Herddb](#), abbiamo configurato 3 nodi zookeeper e 3 nodi Herddb che fanno partire rispettivamente 1 bookies.

7.1 Installazione per dev

Per fare una sperimentazione, bookkeeper ci offre un tool che ci permette di eseguire più bookie in locale. Possiamo dunque eseguire i seguenti passi per testare bookkeeper:

- `mkdir /data/bookie`
- `cd /data/bookie`
- `wget https://bookkeeper-latest-stable-version`
- `tar -xvzf bookkeeper-latest-stable-version.tar.gz`
- `ln -s bookkeeper-latest-stable-version bookkeeper`
- `cd bookkeeper/`

Ora possiamo partire i bookkeeper in locale con:

- `bin/bookkeeper localbookie number_of_bookie_you_want`

Usando bookkeeper visual manager possiamo vedere i bookie che abbiamo fatto partire, in questo caso ne abbiamo fatti partire 6:

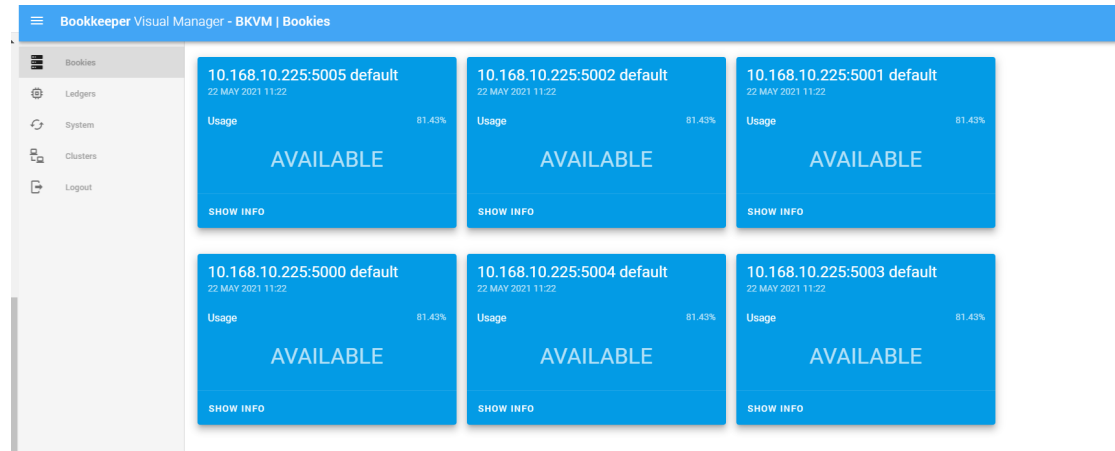


Figure 16: Bookkeeper: localbookie

8 Interazione con Bookkeeper

Bookkeeper offre alcune API per interagire con esso:

- **Ledger API:** Api di basso livello che consente di interagire direttamente con i ledger.
- **Ledger Advanced API:** Un estensione avanzata delle Ledger API per offrire maggior flessibilità alle applicazioni.
- **Distributed log:** API di alto livello, con una semantica simili ai flussi di log continui dal punto di vista delle applicazioni.

Le ledger API consentono di interagire direttamente con i ledgers e quindi consentono di utilizzare bookkeeper in base alle proprie esigenze. In molti casi però, se si desidera uno stream di log o simili, è necessario gestire alcuni aspetti come il monitoraggio della lista dei ledgers, la rotazione dei ledgers e la retention dei dati. In questi casi, la community di bookkeeper consiglia di utilizzare DistributedLog API che offre una astrazione più ad alto livello.

8.1 Ledger API

Per interagire con Bookkeeper usando le api, bisogna innanzitutto inizializzare un oggetto bookkeeper client che necessita della stringa di connessione a zookeeper. In una configurazione di tipo produzione solitamente abbiamo 3 o più nodi di zookeeper. In tal caso bisognerà passare tutta la stringa al client. Quest'ultimo sarà comunque in grado di identificare il nodo leader con cui comunicare.

```

1 try {
2     //The connection string can be also zk1:2181,zk2:2181,zk3:2181
3     String connectionString = "127.0.0.1:2181";
4     BookKeeper bkClient = new BookKeeper(connectionString);
5 } catch (InterruptedException | IOException | KeeperException e) {
6     e.printStackTrace();
7 }

```

Listing 1: Initialize bookkeeper client

La connessione a zookeeper si rende necessaria per diverse ragioni:

- Tutti i metadati dei ledger vengono memorizzati su zookeeper. Quindi il client per la manipolazione dei dati può reperire la lista dei ledger su zookeeper.
- Il client in caso di down di un nodo di bookkeeper o di un cambio ensemble di un ledger sarà in grado di comportarsi di conseguenza in quanto riceverà la notifica da parte di zookeeper.

Prima di inserire delle entries all'interno di Bookkeeper è necessario creare un ledger. Una cosa importante è che se un ledger viene chiuso, questo non può più essere riaperto per la scrittura e bisognerà aprire necessariamente un nuovo ledger.

```

1 byte[] password = "some-password".getBytes();
2 LedgerHandle handle = bkClient.createLedger(ensembleSize, quorumSize,
    ackQuorumSize, BookKeeper.DigestType.MAC, pwd);

```

Listing 2: Creating ledgers

```

1 class LedgerCreationCallback implements AsyncCallback.CreateCallback {
2     public void createComplete(int returnCode, LedgerHandle handle,
3     Object ctx) {
4         System.out.println("Ledger successfully created");
5     }
6 }
7 client.asyncCreateLedger(
8     2,
9     2,
10    2,
11    BookKeeper.DigestType.MAC,
12    password,
13    new LedgerCreationCallback(),
14    "some context"
15 );

```

Listing 3: Creating ledgers asynchronously

Quando si crea un ledger è importante specificare ensembleSize, il quorumSize e ackQuorumSize. Questi parametri se settati nella maniera opportuna garantiscono una replicazione dei dati su più nodi. Questo ci garantisce che in caso di perdita di un nodo del sistema, si avrà la certezza della disponibilità dei dati negli altri nodi. Inoltre se per esempio abbiamo 10 macchine e la replicazione è 3, finché ci saranno abbastanza nodi,

il sistema continuerà a funzionare e il client potrà continuare a leggere e a scrivere. Di base la maggior parte dei sistemi basati su bookkeeper hanno un nodo leader che esegue la manipolazione dei dati (create/delete/update) e gli altri nodi che lavorano come follower leggendo i dati scritti dal leader. In tal caso, i nodi follower dovranno leggere la lista dei ledgers da zookeeper e rimanere in tail sul log delle transazioni di zookeeper per rimanere aggiornati rispetto al leader:

```
1 long entryId = ledger.addEntry("Some entry data".getBytes());
```

Listing 4: Adding entries to ledgers

```
1 Enumerator<LedgerEntry> entries =  
2     handle.readEntries(0, handle.getLastAddConfirmed());  
3  
4 while (entries.hasNextElement()) {  
5     LedgerEntry entry = entries.nextElement();  
6     System.out.println("Successfully read entry " + entry.getId());  
7 }
```

Listing 5: Read entries from the ledger

Quando un nodo scrive su bookkeeper, di base finché non riceve ack la scrittura non viene considerata confermata. Quindi se abbiamo creato il ledger con `ackQuorumSize=2`, per essere sicuri che la scritta sia andata a buon fine dovremo aspettare di ricevere almeno 2 ack. Bookkeeper però permette anche di leggere il valore delle entries che non hanno ancora il `LastAddConfirmed`. Questo chiaramente potrebbe portare a problemi di consistenza e sarebbe da usare solo nel caso in cui non abbiamo la necessità di garantire consistenza tra i dati.

```
1 Enumerator<LedgerEntry> entries =  
2     handle.readUnconfirmedEntries(0, lastEntryIdExpectedToRead);  
3  
4 while (entries.hasNextElement()) {  
5     LedgerEntry entry = entries.nextElement();  
6     System.out.println("Successfully read entry " + entry.getId());  
7 }
```

Listing 6: Reading entries after the `LastAddConfirmed` range

Cancellazione di un ledger:

```
1 long ledgerId = 1234;  
2  
3 try {  
4     bkClient.deleteLedger(ledgerId);  
5 } catch (Exception e) {  
6     e.printStackTrace();  
7 }  
8 }
```

Listing 7: Deleting ledgers

```
1 class DeleteEntryCallback implements AsyncCallback.DeleteCallback {  
2     public void deleteComplete() {
```

```

3         System.out.println("Delete completed");
4     }
5 }
6 bkClient.asyncDeleteLedger(ledgerID, new DeleteEntryCallback(), null);

```

Listing 8: Deleting ledgers asynchronously

Esempio:

```

1     String connectionString = "10.168.10.225:2181";
2     try {
3         //Initialize connection
4         BookKeeper bkClient = new BookKeeper(connectionString);
5         byte [] pwd = "test".getBytes();
6         //Create ledger
7         /**
8          * First parameter: ensemble size
9          * Secondo parameter: quorumSize
10          * Third parameter: Ack quorum size
11          */
12         LedgerHandle handle = bkClient.createLedger(2,2,2, BookKeeper.
DigestType.MAC, pwd);
13         //Print ledger metadata
14         System.out.println("Ledger metadata: " + handle.getLedgerMetadata
());
15
16     } catch (IOException | InterruptedException | BKException e) {
17         e.printStackTrace();
18     }

```

Listing 9: Creating ledger example

In questo esempio abbiamo creato un ledger, con ensembleSize=2, quorumSize=2 e ack-QuorumSize=2. Dunque Bookkeeper in questo caso, formerà un ensemble composto da due nodi. Il ledger verrà quindi replicato sull'ensemble che è stato scelto. Quando viene scritto una voce su quel ledger, il client si aspetterà di avere almeno due ack per considerare la scrittura andata a buon fine:

```

2021-05-22 12:18:07,265 - INFO - [main-EventThread:LedgerCreateOp@254] -
↪ Ensemble: [10.168.10.225:5003, 10.168.10.225:5005] for ledger: 1

```

```

Ledger metadata: LedgerMetadata{formatVersion=3, ensembleSize=2,
↪ writeQuorumSize=2, ackQuorumSize=2, state=OPEN, digestType=MAC,
↪ password=base64:dGVzdA==, ensembles={0=[10.168.10.225:5003,
↪ 10.168.10.225:5005]}, customMetadata={}}

```

Su bookkeeper visual manager:

Min Length	Max Length	0

Ledger 1

Cluster: default
Age: 1 minutes
Created at: 22 MAY 2021 12:19
State: OPEN
Ensemble size: 2 (number of bookies)
WriteQuorumSize: 2 (number of copies)
AckQuorumSize: 2
Size: 0 Bytes
LastEntryId: -1
Password: test
DigestType: MAC
Closed: false
MetadataFormatVersion: 3

Ensemble 0: 10.168.10.225:5003; 10.168.10.225:5005;

Figure 17: Bookkeeper: Ledger 1

Esempio 2:

```

1      String connectionString = "10.168.10.225:2181";
2      try {
3          //Initialize connection
4          BookKeeper bkClient = new BookKeeper(connectionString);
5
6
7          byte [] pwd = "test".getBytes();
8          //Create ledger
9          /**
10         * First parameter: ensemble size
11         * Secondo parameter: quorumSize
12         * Third parameter: Ack quorum size
13         */
14         LedgerHandle lh = bkClient.createLedger(2,2,2, BookKeeper.
DigestType.MAC, pwd);
15         //Print ledger metadata
16         System.out.println("Ledger metadata: " + lh.getLedgerMetadata
());
17
18         long ledgerId = lh.getId();
19
20         // Create a buffer for four-byte entries
21         ByteBuffer entry = ByteBuffer.allocate(4);
22

```

```

23         int numberOfEntries = 100;
24
25         // Add entries to the ledger, then close it
26         for (int i = 0; i < numberOfEntries; i++){
27             entry.putInt(i);
28             entry.position(0);
29             lh.addEntry(entry.array());
30         }
31         lh.close();
32         // Open the ledger for reading
33         lh = bkClient.openLedger(ledgerId, BookKeeper.DigestType.MAC,
    pwd);
34
35         // Read all available entries
36         Enumeration<LedgerEntry> entries = lh.readEntries(1,
    numberOfEntries - 1);
37
38         while(entries.hasMoreElements()) {
39             ByteBuffer result = ByteBuffer.wrap(entries.nextElement()
    .getEntry());
40             Integer retrEntry = result.getInt();
41
42             // Print the integer stored in each entry
43             System.out.println(String.format("Result: %s", retrEntry)
    );
44         }
45
46         // Close the ledger and the client
47         lh.close();
48         bkClient.close();
49
50     } catch (IOException | InterruptedException | BKException e) {
51         e.printStackTrace();
52     }
53 }

```

Listing 10: Creating and read on leader

Dai log possiamo vedere:

```

2021-05-22 12:51:39,915 - INFO - [main-EventThread:LedgerCreateOp@254] -
    ↳ Ensemble: [10.168.10.225:5000, 10.168.10.225:5005] for ledger: 2
Ledger metadata: LedgerMetadata{formatVersion=3, ensembleSize=2,
    ↳ writeQuorumSize=2, ackQuorumSize=2, state=OPEN, digestType=MAC,
    ↳ password=base64:dGVzdA==, ensembles={0=[10.168.10.225:5000,
    ↳ 10.168.10.225:5005]}, customMetadata={}}
2021-05-22 12:51:41,759 - INFO -
    ↳ [bookkeeper-io-3-1:PerChannelBookieClient$ConnectionFutureListener@2463]
    ↳ - Successfully connected to bookie: 10.168.10.225:5000 [id:
    ↳ 0x0ef4bf90, L:/192.168.51.11:60399 - R:/10.168.10.225:5000]

```

```

2021-05-22 12:51:41,759 - INFO -
↳ [bookkeeper-io-3-2:PerChannelBookieClient$ConnectionFutureListener@2463]
↳ - Successfully connected to bookie: 10.168.10.225:5005 [id:
↳ 0x3297c08c, L:/192.168.51.11:60400 - R:/10.168.10.225:5005]
2021-05-22 12:51:41,783 - INFO -
↳ [bookkeeper-io-3-1:PerChannelBookieClient$1@491] - connection [id:
↳ 0x0ef4bf90, L:/192.168.51.11:60399 - R:/10.168.10.225:5000]
↳ authenticated as BookKeeperPrincipal{ANONYMOUS}
2021-05-22 12:51:41,783 - INFO -
↳ [bookkeeper-io-3-2:PerChannelBookieClient$1@491] - connection [id:
↳ 0x3297c08c, L:/192.168.51.11:60400 - R:/10.168.10.225:5005]
↳ authenticated as BookKeeperPrincipal{ANONYMOUS}
Result: 1
Result: 2
Result: 3
Result: 4
Result: 5
Result: 6
Result: 7
#ect

```

Se guardiamo su Bookkeeper visual manager vedremo che **LastEntryId=99** dato che abbiamo inserito 99 entries all'interno del ledger 2.

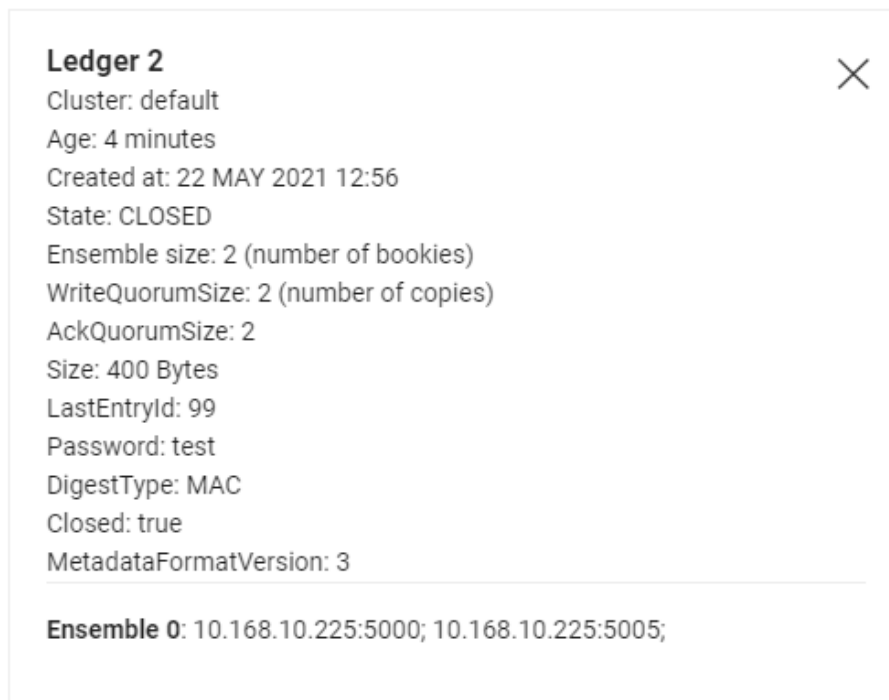


Figure 18: Bookkeeper: Ledger 2 with entries

Come vediamo il concetto è molto semplice, ovvero:

- Creiamo un ledger passando i parametri di replicazione opportuni
- Inseriamo le entries all'interno del ledger
- Leggiamo le entries all'interno del ledger

8.2 Ledger Advanced API

Le API avanzate sono semplicemente un'estensione della versione base, con la differenza che si può passare un entryid nella fase di inserimento di una entry. E' inoltre possibile creare un ledger passandogli già l'id del ledger. Questa soluzione implica chiaramente un maggior controllo, in quanto se il ledger esiste già viene generata un'eccezione **LedgerExistsException**. Alcune note che bisogna prendere in considerazione quando si usano queste API sono:

- L'entryid non deve essere un valore negativo.
- I client possono aggiungere le entries in ordine sparso.
- Tuttavia le entries vengono riconosciute in ordine a partire da 0.

Per la creazione di un ledger:

```
1 byte[] passwd = "some-passwd".getBytes();
2 LedgerHandleAdv handle = bkClient.createLedgerAdv(
3     3, 3, 2, // replica settings
4     DigestType.CRC32,
5     passwd);
6 }
```

Listing 11: Creating ledger

Se vogliamo invece specificare manualmente il ledger-id:

```
1 long ledgerId = ...; // the ledger id is generated externally.
2
3 byte[] passwd = "some-passwd".getBytes();
4 LedgerHandleAdv handle = bkClient.createLedgerAdv(
5     ledgerId, // ledger id generated externally
6     3, 3, 2, // replica settings
7     DigestType.CRC32,
8     passwd);
```

Listing 12: Creating ledger with custom ledgerid

Per l'aggiunta di una entry specificando id:

```
1 long entryId = ...; // entry id generated externally
2
3 ledger.addEntry(entryId, "Some entry data".getBytes());
```

Listing 13: Add entry with custom entryid

9 DistributedLog (DL)

L'Api distributed log è un servizio di replicazione dei log che permette di utilizzare Bookkeeper senza la necessità di interagire direttamente con i ledger. DL conserva le sequenze di records in categorie chiamate logs (aka logs streams). Abbiamo due processi in DL:

- **Writer:** che scrivono i record in un log DL
- **Reader:** che leggono dai log e elaborano i record

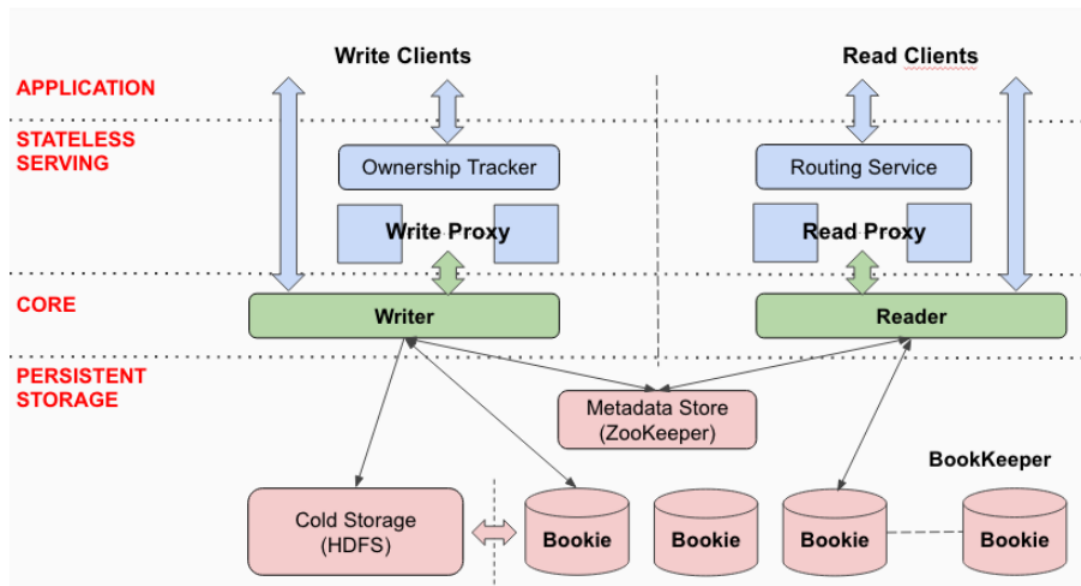


Figure 19: Bookkeeper: DistributedLog architecture

9.1 Logs

Un log in DL è una sequenza ordinata, immutabile di records.

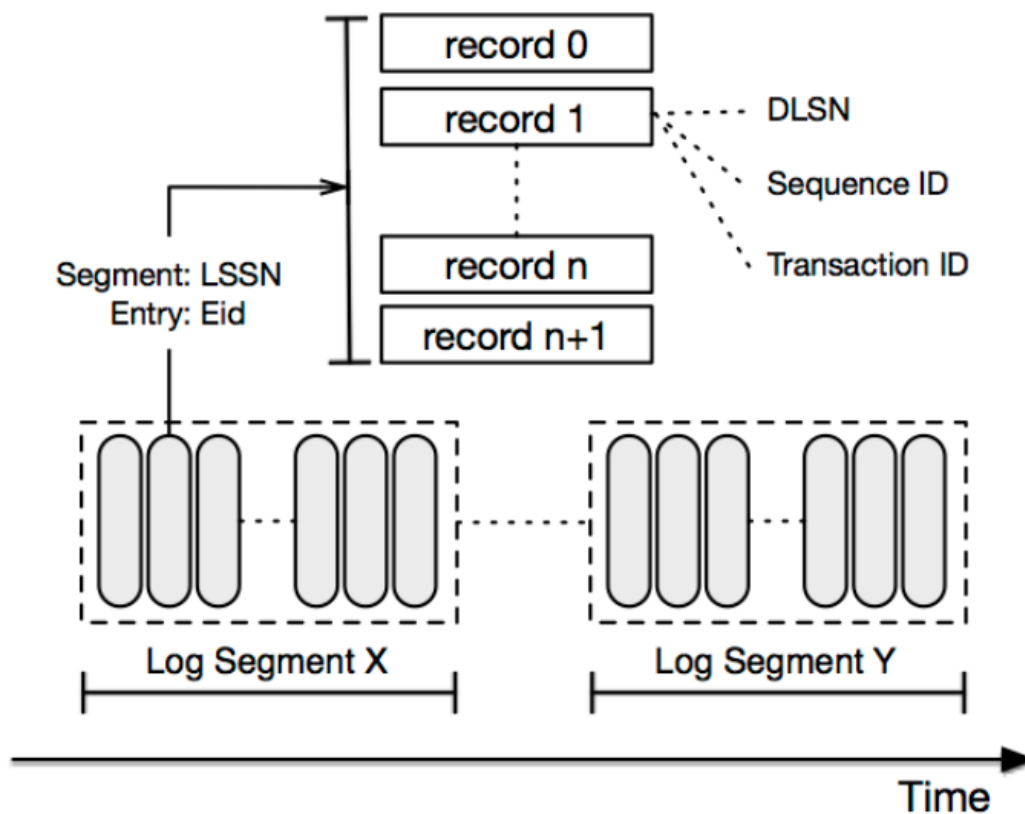


Figure 20: Bookkeeper: DistributedLog Log diagram

9.1.1 Log records

Ogni record all'interno di un log è una sequenza di byte. Le applicazioni sono responsabili della serializzazione e della de-serializzazione delle sequenze di byte archiviate nei log. I record vengono scritti in sequenza in un flusso di log e assegnati con un numero di sequenza univoco chiamato **DLSN**(Distributed Log Sequence Number). Le applicazioni possono inoltre assegnare un proprio numero di sequenza durante la creazione dei **log records**. Questi numeri di sequenza sono noti come **TransactionIDs** o **txid**. DLSN o TransactionID possono essere utili per posizionare i **reader** per avviare la lettura da un record specifico all'interno del log.

9.1.2 Log segments

Ogni log viene suddiviso in **log segments** che contengono i sottoinsiemi di records. I log segments vengono poi distribuiti e archiviati in Bookkeeper. DL esegue la rotazione dei log segments in base alla policy di rotazione configurata, che può essere:

- Un periodo di tempo configurabile
- Una dimensione massima configurabile

I dati all'interno dei log vengono suddivisi in log segments di uguale dimensione e vengono distribuiti uniformemente tra i vari bookies. questo consente ai logs di scalare rispetto alla dimensione del singolo bookie e di distribuire il carico di lettura in tutto il cluster.

9.1.3 Namespace

Gli streams di logs che appartengono alla stessa organizzazione sono generalmente classificati e gestiti come namespace. I namespace in DL consentono alle applicazioni di individuare streams di logs. Le applicazioni in un namespace possono:

- Create streams
- Eliminare streams
- Troncare gli streams in un determinato numero di sequenza (DLSN o un TransactionID)

Vedi anche [Namespace](#).

9.1.4 Writer

Tramite l'api di DL, i writer scrivono i dati nei logs a loro scelta. Tutti i records vengono scritti nei logs in ordine. La sequenza è definita dallo scrittore, il che significa che c'è solo uno scrittore attivo per un log in un dato momento. DL garantisce la correttezza quando due writer tentano di scrivere nello stesso log nel momento in cui si verifica una partizione di rete utilizzando un meccanismo di fencing nell'archivio dei log segments. I writer vengono serviti e gestiti in un livello di servizio chiamato **Write proxy**. E' usato principalmente per la gestione di molteplici scritture da un numero elevato di clients.

9.1.5 Reader

I readers in DL leggono i record dai logs a loro scelta, a partire da una posizione fornita che può essere un DLSN o un TransactionID. Leggono i records dai logs in un ordine ben preciso e diversi readers possono leggere da differenti posizioni nello stesso log. Per avere buone prestazioni in lettura, i log records possono essere memorizzati nella cache in un livello di servizio chiamato **Read proxy**. Usato per la gestione di molteplici lettura da un numero elevato di clients.

10 Sicurezza

La comunicazione tra vari client se non diversamente configurati, solitamente avviene in maniera non protetta. Se si ha un ambiente che gestisce una grande quantità di dati che potrebbero essere sensibili o banalmente essere di forte impatto nel proprio

business aziendale qualora venissero danneggiati, è opportuno che l'accesso a tali dati sia possibile solo a client autenticati o fidati. In merito a questo Apache Bookkeeper permette di configurare un'autenticazione di sito SASL basato su Kerberos. Questo permette eventualmente al server bookkeeper di accettare solo connessioni da client che abbiano un ticket valido per quell'ambiente. Inoltre bookkeeper permette di avere un trasferimento di dati in maniera criptato tra il client e il server bookkeeper tramite il TLS.

10.1 TLS

E' possibile configurare i bookie che fanno parte del cluster per comunicare in maniera criptata utilizzando un certificato ssl con la corrispondente chiave. Per prendere ancora più sicuro le varie comunicazioni è possibile configurare un truststore contenente tutti i certificati. Questo permette ai bookie di fidarsi solo dei certificati ssl che sono presenti all'interno del truststore. Bisognerà dunque configurare sia il client che il server dopo aver generato i corrispondenti certificati e chiavi.

Lato server bisognerà settare i seguenti parametri per ogni bookie dell'ensemble:

```
tlsProvider=OpenSSL
# key store
tlsKeyStoreType=JKS
tlsKeyStore=/var/private/tls/bookie.keystore.jks
tlsKeyStorePasswordPath=/var/private/tls/bookie.keystore.passwd
# trust store
tlsTrustStoreType=JKS
tlsTrustStore=/var/private/tls/bookie.truststore.jks
tlsTrustStorePasswordPath=/var/private/tls/bookie.truststore.passwd
```

Oltre a queste impostazioni è possibile abilitare alcune opzioni che sono di base opzionali quali:

- **tlsClientAuthentication:** Può essere true/false e se abilitata autentica anche l'altra estremità del canale di comunicazione. Dunque se questa opzione è a true, anche lato client bisognerà settarlo a true.
- **tlsEnabledCipherSuites:** Con questa opzione è possibile specificare la lista degli algoritmi per lo scambio chiavi da utilizzare per negoziare le impostazioni di sicurezza di una connessione di rete basato su TLS.
- **tlsEnabledProtocols:** Con questa opzione è possibile specificare la lista dei protocolli TLS che il server deve accettare dai client. Es. TLSv1.2,TLSv1.1,TLSv1

I client che si vorranno connettere al server, dovranno necessariamente avere un certificato valido:

```
tlsProvider=OpenSSL
clientTrustStore=/var/private/tls/client.truststore.jks
clientTrustStorePasswordPath=/var/private/tls/client.truststore.passwd
```

Nella configurazione qua sopra può essere usata qualora non fosse richiesta l'autenticazione del client. Ovvero se lato server non fosse settato l'opzione **tlsClientAuthentication=true**. In caso contrario la configurazione diventerebbe la seguente:

```
tlsClientAuthentication=true
clientKeyStore=/var/private/tls/client.keystore.jks
clientKeyStorePasswordPath=/var/private/tls/client.keystore.passwd
```

10.2 Kerberos

Kerberos è un protocollo di rete per l'autenticazione, progettato per fornire un'autenticazione avanzata per le applicazioni client/server utilizzando la crittografia a chiave segreta.

In uno scenario standard, la maggior parte dei sistemi utilizza un'autenticazione basata su username e password. Quando l'utente effettua un'autenticazione presso un server, deve fornire un nome utente e una password. Queste informazioni sfortunatamente per molti servizi viaggiano in maniera non criptata. Questo scenario non comporta pericoli qualora il sistema in questione si trovi in una rete locale, non sia visibile all'esterno o gli utenti siano fidati.

Tuttavia se colleghiamo questo sistema ad internet, non si potrà più assumere che la rete sia sicura. Qualunque aggressore che ha accesso alla rete e che può utilizzare un analizzatore di pacchetti di rete (packet sniffer), può intercettare la password e i nomi utente che lo attraversano, compromettendo così gli account dell'utente e l'integrità dell'infrastruttura. Lo scopo principale di Kerberos è quello di eliminare la trasmissione delle informazioni di autenticazione attraverso la rete. La sua sicurezza è basata fortemente sui timestamp e sui tempi di vita, usati come indicatori per la creazione di una comunicazione sicura e per evitare attacchi.

10.2.1 Componenti

- **Authentication server (AS)** mantiene il database delle chiavi segrete. Il client e il server condividono la chiave solo con AS. Quindi il possesso non autorizzato delle credenziali di accesso all'AS compromette la sicurezza dell'architettura intera.
- **Ticket Granting Service (TGS)** è il server che fornisce il ticket per autenticazione presso SS. Il ticket ha una validità e se scade deve essere rinnovato.
- **Ticket Granting Ticket (TGT)** è il ticket che viene fornito al client dal AS.
- **Service Server (SS)** il server che fornisce il servizio.
- **Key Distributed Center (KDC)** comprende AS, TGS e database del principal con le chiavi criptate

10.2.2 Protocollo

- **Utente: Autenticazione base:**
L'utente inserisce username e password

- **Client: Autenticazione AS:**

1. Il client manda un messaggio non criptato all'AS richiedendo servizi per l'utente. Non vengono inviati nè la chiave segreta nè la password.
2. L'AS controlla se il client è nel suo database, Se lo è invia due messaggi al client:
 - **Messaggio A:** Chiave di sessione client-TGS criptata usando la chiave segreta dell'utente.
 - **Messaggio B:** Ticket-Granting Ticket contenente l'identificativo del client, l'indirizzo di rete, il tempo di validità del ticket e la chiave di sessione client-TGS. Il TGT è criptato utilizzando la chiave segreta di TGS.
3. Quando il client riceve il messaggio A e B, decripta il messaggio A ottenendo la chiave di sessione client-TGS. Questa chiave è utilizzata per le successive comunicazioni con TGS. Il client non può decriptare il messaggio B perché quest'ultima è criptata con la chiave segreta TGS. A questo punto il client con la chiave di sessione può autenticarsi presso TGS.

- **Client: Autenticazione TGS:**

1. Quando richiede servizi, il client invia due messaggi a TGS:
 - **Messaggio C:** Composto dal TGT (Contenuto del messaggio B) e dal identificativo del servizio richiesto.
 - **Messaggio D:** Authenticator (formato da identificativo del client e timestamp), criptato usando la chiave di sessione del client-TGS.
2. Ricevendo i messaggi C e D, TGS decripta il messaggio C con la propria chiave e dal messaggio estrae la chiave di sessione client-TGS che utilizza per decriptare il messaggio D. A questo punto manda due messaggi al client:
 - **Messaggio E:** Ticket client-server che include l'identificativo del client, l'indirizzo di rete del client, il periodo di validità e la chiave di sessione client-server. Il messaggio viene criptato utilizzando la chiave segreta del server che offre il servizio.
 - **Messaggio F:** Chiave di sessione client-server criptato utilizzando la chiave di sessione client-TGS.

- **Client: Autenticazione SS:**

1. Ricevendo i messaggi E e F dal TGS, il client può autenticarsi presso il SS. Il client si connette al SS e invia due messaggi:
 - **Messaggio E:** Ticket client-server criptato usando la chiave segreta di SS.

- **Messaggio G:** Un nuovo Authenticator, che include l'identificativo del client, il timestamp ed è criptato usando la chiave di sessione client-server.
- 2. Il server decripta il ticket usando la sua chiave segreta e invia un messaggio al client per confermare la propria identità e la volontà di fornire il servizio al client:
 - **Messaggio H:** Il timestamp trovato nell'autenticazione incrementato di uno, criptato utilizzando la chiave di sessione client-server.
- 3. Il client decripta la conferma usando la chiave di sessione client-server e controlla che il timestamp sia correttamente aggiornato. Se lo è, il client può considerare affidabile il server e iniziare a effettuare le richieste di servizio.
- 4. Il server fornisce i servizi al client.

10.2.3 Configurazione kerberos su Bookkeeper

Per la configurazione è necessario prima di tutto avere un'installazione Kerberos server, nella quale andremo a definire i vari principal ed a generare i keytab necessari. Sarebbe opportuno per avere una forte sicurezza, kerberizzare oltre alla comunicazione tra client e bookkeeper anche da bookkeeper a zookeeper.

```
#On kerberos server
#Add principal
kadmin: addprinc -randkey zookeeper/{hostname}@{REALM}
kadmin: addprinc -randkey bookkeeper/{hostname}@{REALM}
#For client it's better create an headless principal
kadmin: addprinc -randkey bookkeeper@{REALM}
#Generate keytab files
kadmin: ktadd -k /keytab_path/{keytabname}.keytab
↪ zookeeper/{hostname}@{REALM}
kadmin: ktadd -k /keytab_path/{keytabname}.keytab
↪ bookkeeper/{hostname}@{REALM}
kadmin: ktadd -k /keytab_path/{keytabname}.keytab bookkeeper@{REALM}

#Create krb5.conf file containing kdc
[libdefaults]
    default_realm = EXAMPLE.COM
    dns_lookup_kdc = true
    dns_lookup_realm = false
    allow_weak_crypto = true

[realms]
    EXAMPLE.COM = {
        kdc = kerberos.example.com
```

```

        kdc = kerberos-1.example.com
    }

[domain_realm]
    .example.com = EXAMPLE.COM
    example.com = EXAMPLE.COM

#Enable kerberos on zookeeper server
#In zoo.cfg file
authProvider.1 =
    ↪ org.apache.zookeeper.server.auth.SASLAuthenticationProvider
kerberos.removeHostFromPrincipal = true
kerberos.removeRealmFromPrincipal = true
#Create jaas.conf on zookeeper root path
#Add this config
Server {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true keyTab="/keytab_path/zookeeper.keytab"
    storeKey=true
    useTicketCache=false
    principal="zookeeper/{hostname}@{REALM}";
};
#Create java.env file in zookeeper conf directory
#and paste this config
export JVMFLAGS=" -Djava.security.krb5.conf=krb5.conf
    ↪ -Djava.security.auth.login.config=jaas.conf"

#start server
bin/zkServer.sh start conf/zoo.cfg

#On bookkeeper servers
#Create jaas.conf file containing this config
Bookie {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/keytab_path/bookie.keytab"
    principal="bookkeeper/{hostname}@{REALM}";
};
// ZooKeeper client authentication
Client {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true

```

```

    storeKey=true
    keyTab="/keytab_path/bookie.keytab"
    principal="bookkeeper/{hostname}@{REALM}";
};

#Pass the name of the JAAS file as a JVM parameter to each Bookie and
↪ specify krb5.conf
-Djava.security.auth.login.config=jaas.conf
↪ -Djava.security.krb5.conf=krb5.conf

#Enable SASL authentication plugin in the bookies by setting following
↪ parameter
bookieAuthProviderFactoryClass=org.apache.bookkeeper.sasl.SASLBookieAuthProviderFactory

#On bookkeeper clients
#Create jaas.conf containing this config
BookKeeper {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/keytab_path/bookkeeper.keytab"
    principal="bookkeeper-client-1@{REALM}";
};

#Pass the name of the JAAS file as a JVM parameter to the client JVM and
↪ set specify krb5.conf
-Djava.security.auth.login.config=jaas.conf
↪ -Djava.security.krb5.conf=krb5.conf

#Enable SASL authentication plugin in the client by setting following
↪ parameters.
clientAuthProviderFactoryClass=org.apache.bookkeeper.sasl.SASLClientProviderFactory

```

11 Demo

La [demo](#) che è stato sviluppato illustra come funziona in maniera basilare come può essere sfruttare bookkeeper. Tramite il client di bookkeeper, un nodo può creare un ledger e iniziare a scriverci. Parallelamente altri nodi potrebbero mettersi a leggere le entry che sono stati scritti nel ledger. Nella nostra demo facciamo una semplice leadership election in modo tale da avere un solo nodo che scrivere. Tale nodo viene denominato nodo leader. Gli altri nodi che non posso scrivere leggono le entry che sono stati scritti dal leader. Quando il nodo leader parte, riceve come paramenti ensembleSize, writeQuorumSize e ackQuorumSize. Questi parametri vengono utilizzati per definire, quanti nodi tenere in considerazione per ensemble; Su quanti nodi replicare le entry e quanti ack

bisogna aspettare per considerare la scrittura andata a buon fine. Questo permette di avere una grossa disponibilità in quando se partissimo per esempio con 2,2,2 verrebbero scelti due nodi come ensemble e replicato le scritture su due nodi. Qualora perdessimo un nodo, avremo comunque sempre l'altro nodo che conterrà tutte le entry. Più ovviamente il numero di bookie aumenta e l'ensembleSize diventa più grande, maggiore sarà la disponibilità che potremo avere. Bisognerà tenere conto anche della latenza di rete per le scritture e le relative ACK. Con Bookkeeper è possibile sviluppare sistemi molto più complessi con una alta disponibilità come HerdDB o Pulsar. Essendo un sistema di basso livello sviluppare una applicazione basata su bookkeeper richiede una buona conoscenza del sistema. Rispetto alla demo su cui ci siamo ispirati, le differenze fondamentali sono:

- Il sistema può essere avviato con un zookeeper e un bookkeeper embedded senza la necessità di installare un cluster manualmente.
- Quando si avvia il sistema si può specificare che i parametri ensembleSize, ackQuorumSize, writeQuorumSize.
- Il sistema ha solo un nodo che scrive e gli altri nodi leggono rimanendo in tail.
- La scelta del leader viene fatto attraverso il framework curator che tramite zookeeper decide qual'è il nodo designato per la scrittura.

12 Conclusione

Apache Bookkeeper è un software a cui ho contribuito durante le ore lavorative in quanto è uno dei componenti core di uno dei nostri sistemi. E' un sistema molto complesso in quanto di basso livello, di conseguenza difficile da debuggare in caso di problemi. In base alla mia esperienza, amministrare bookkeeper risulta in generale molto complesso, ma presenta una comunità molto forte e attiva che è sempre pronto a rispondere e risolvere i problemi che si presentano.

References

- [1] Apache Bookkeeper. A scalable, fault-tolerant, and low-latency storage service optimized for real-time workloads.
- [2] Enrico Olivelli. How to build a distributed database with apache bookkeeper.
- [3] Sijia-w. Introduzione ad apache bookkeeper.
- [4] Sijia-w. Why apache bookkeeper? consistency, durability, availability.
- [5] Leigh Stewart. Building distributedlog: High-performance replicated log service.